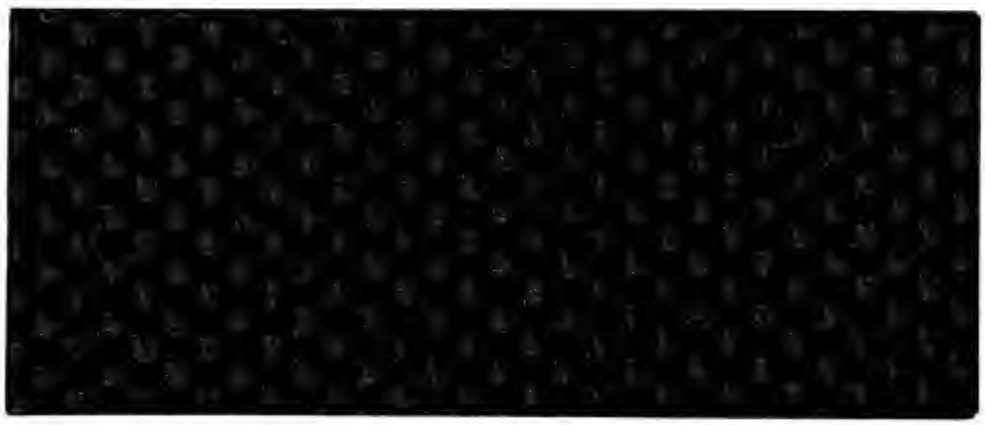
Faster Optimal Parallel Prefix Sums and List Ranking

by

Richard Cole†
Uzi Vishkin‡

Ultracomputer Note #117
Computer Science Technical Report #277
February, 1987



Faster Optimal Parallel Prefix Sums and List Ranking

by

Richard Cole†
Uzi Vishkin‡

Ultracomputer Note #117
Computer Science Technical Report #277
February, 1987

† This research was supported in part by NSF grant DCR-84-01633, ONR grant N00014-85-K-0046 and by an IBM faculty development award.

‡ This research was supported in part by NSF grants NSF-CCR-8615337 and NSF-DCR-8413359, ONR grant N00014-85-K-0046 and by the Applied Mathematical Science subprogram of the office of Energy Research, U.S. Department of Energy under contract number DE-AC02-76ER03077. **Current Address:** Tel Aviv University, Department of Computer Science, School of Mathematical Science, Tel Aviv 69 978, Israel

ABSTRACT

We present a parallel algorithm for the prefix sums problem which runs in time $O(\log n / \log\log n)$ using $n\log\log n / \log n$ processors (optimal speed up). This algorithm leads to a parallel list ranking algorithm which runs in $O(\log n)$ time using $n/\log n$ processors (optimal speed up).

1. Introduction

The model of parallel computation used in this paper is the concurrent-read concurrent-write (CRCW) parallel random access machine (PRAM). A PRAM employs p synchronous processors all having access to a common memory. A CRCW PRAM allows simultaneous access by more than one processor for read and write purposes. Here we use a version of the CRCW PRAM which allows an attempt by several processors to write into the same memory location provided that they are trying to write the same value. In such a case this value will be written into this memory location in $O(1)$ time. See [V-83] for a survey of results concerning PRAMs.

Let $Seq(n)$ be the fastest known worst-case running time of a sequential algorithm, where n is the length of the input for the problem being considered. Obviously, the best upper bound on the parallel time achievable using p processors, without improving the sequential result, is of the form $O(Seq(n)/p)$. A parallel algorithm that achieves this running time is said to have *optimal speed-up* or more simply to be *optimal*.

In the present paper we consider three problems:

1. *Input.* An array of n numbers $A(1),A(2),..,A(n)$.

   The prefix sums problem. Compute $\sum_{j=1}^{i} A(j)$ for all $1 \leq i \leq n$.

2. *Input.* A connected directed graph $G(V,E)$. The in-degree of each vertex is exactly one. The out-degree of each vertex is exactly one. Note that the graph forms a directed circuit. The vertices are given in an array of size n. Each vertex has a pointer to the next vertex (representing its outgoing edge). We define a subset U of V to be an *$r-$ruling set* of G if:

   (1) No two vertices of U are adjacent.

   (2) For each vertex v in V there is a directed path from v to some vertex in U whose edge length is at most r.

   The r-ruling set problem. Find an r-ruling set of V.

3. *Input.* A linked list of length n. It is given in an array of length n, not necessarily in the order of the linked list. Each of the n elements (except the last element in the

linked list) has the array index of its successor in the linked list.

The list ranking problem. For each element, compute the number of elements following it in the linked list.

Section 2 presents a new algorithm which solves the prefix sums problem in $O(\log n / \log\log n)$ time using $n\log\log n / \log n$ processors (optimal speed up), provided that each $A(i)$ is represented by $O(\log n)$ bits. [FL-80] presented the standard parallel algorithm for the prefix sums problem (see also Section 2.1). It runs in $O(\log n)$ time using $n/\log n$ processors and is free of read or write conflicts. This algorithm has become one of the most heavily used routines in parallel algorithms. Therefore the new algorithm is of interest beyond its application to the list ranking problem as presented in this paper. Our algorithm improves a similar result due to Reif [R-85].

Sections 3 and 4 are a follow-up of the paper [CV-86a]. The reader may find it helpful to read sections 2, 3 and 4 of this early paper before reading sections 3 and 4 here, although the presentation in the present paper is self contained. There, we gave a parallel algorithm for the 2-ruling set problem which runs in $O(\log n)$ time using $n/\log n$ processors without read or write conflicts. Section 3 gives a parallel algorithm for this problem which runs in $O(\log n / \log\log n)$ time using $n\log\log n / \log n$ processors (optimal speed up), using the new prefix sums algorithm. [W-79] gave a parallel algorithm for the list ranking problem which runs in $O(\log n)$ time and uses n processors (see also Section 4.1). This simple algorithm is sometimes referred to as the standard parallel list ranking algorithm. Wyllie conjectured that it is impossible to design a logarithmic time algorithm for this problem which achieves optimal speed-up. [CV-86a] introduced a novel technique called *deterministic coin tossing* which led to efficient parallel algorithms for the r-ruling set problems for various values for r (see also Section 3.1). This technique is the backbone of all available poly-log time optimal (deterministic) parallel algorithms for the list ranking problem. The fastest optimal parallel list ranking algorithm in [CV-86a] runs in $O(\log n \log^* n)$ time. It is without read or write conflicts. Section 4 gives a parallel list ranking algorithm which runs in $O(\log n)$ time using $n/\log n$ processors, using the new 2-ruling set algorithm. Recently, [CV-86b] were able to invalidate Wyllie's conjecture by giving the first parallel algorithm for the list ranking problem which runs in $O(\log n)$ time using $n/\log n$ processors. It is based on a general method for assigning processors to jobs (called approximate task scheduling) which uses expander graphs. The algorithm presented here is considerably simpler and its time bounds have small constants. The other algorithm, however, is free of read and write conflicts. [AM-86] gave recently another logarithmic time optimal parallel list ranking

algorithm, again without read and write conflicts. It is interesting to mention that before the deterministic coin tossing technique was known, [V-84] introduced the idea of using randomization in order to obtain an optimal speed-up parallel algorithm for list ranking. Specifically, the fastest optimal parallel list ranking algorithm in [V-84] runs in $O(\log n \log^* n)$ time with overwhelming probability. It is without read or write conflicts. A very simple optimal parallel randomized algorithm which runs in $O(\log n \log\log n)$ time was also given in [V-84]. A step of this simple algorithm consists of applying the standard logarithmic time optimal parallel prefix sums algorithm $O(\log\log n)$ times. The final revision of the journal version of [V-84] was done after the parallel prefix sums of the present paper was found. This journal version explains how to simply replace the standard parallel prefix sums algorithm by the new parallel prefix sums algorithm of the present paper in order to get logarithmic time and optimal speed up. (Actually, there is a strong similarity between the framework of this simple randomized algorithm and the deterministic list ranking algorithm of the present paper.) Miller and Reif provide another logarithmic time randomized algorithm for list ranking [MR-85]. Yet another simple logarithmic time randomized list ranking was given in [AM-86]. It is without read or write conflicts.

The list ranking problem is often encountered in the design of parallel algorithms. For instance the fundamental techniques for parallel algorithm on trees (the "Euler tour technique" of [TV-85] and [V-85] and the "accelerated centroid decomposition" technique of [CV-86c], have the same complexity as the new list ranking algorithm presented here. The logarithmic time optimal parallel list ranking has also led to achieving such efficiencies for several graph problems, including connectivity [CV-86b]. For more on this see [CV-86b].

2. Fast Parallel Prefix Sums

2.1. Preliminaries

Theorem (Brent). Any synchronous parallel algorithm taking time t that consists of a total of x elementary operations can be implemented by p processors within a time of $\lfloor x/p \rfloor + t$.

Proof of Brent's theorem. Let x_i denote the number of operations performed by the algorithm in time i ($\sum_1^t x_i = x$). We use the p processors to "simulate" the algorithm. Since all the operations at time i can be executed simultaneously, they can be computed by the p processors in $\lceil x_i/p \rceil$ units of time. Thus, the whole algorithm can be implemented by p processors in time

$$\sum_1^t \lceil x_i / p \rceil \le \sum_1^t (\lfloor x_i / p \rfloor + 1) \le \lfloor x/p \rfloor + t \ .\square$$

Remark 2.1.1. Brent's theorem is stated for models of computation where not all computational overheads are taken into account. Specifically, the proof of Brent's theorem poses two implementation problems. The first is to evaluate x_i at the beginning of time i in the algorithm. The second is to assign the processors to their jobs.

Remark 2.1.2. Whenever in the present paper the implementation problems as per Remark 2.1.1 can be readily overcome, we allowed ourselves to switch freely from results of the form "$O(x)$ operations and $O(t)$ time" to result of the form "x/t processors and $O(t)$ time," or "$O(x/p + t)$ time using p processors." Sometimes we do it without even mentioning that Brent's theorem is used.

We briefly review the standard prefix sums algorithm. We describe the algorithm recursively:

PREFIX_SUMS$(A(1),A(2),\ldots,A(n))$:

PREFIX_SUMS$(A(1)+A(2),A(3)+A(4),\ldots,A(2i-1)+A(2i),\ldots)$

In one parallel step compute $\sum_{j=1}^{i} A(j)$ for each odd i, $1 \le i \le n$.

The depth of the recursion is $\lceil \log n \rceil$. [1] Recursive call k needs $O(n/2^k)$ operations and $O(1)$ time. Therefore, the algorithm needs $O(n)$ operations and $O(\log n)$ time. It is easy to implement this algorithm to run in $O(\log n)$ time using $n/\log n$ processors, using Brent's theorem.

2.2. The New Algorithm

Our algorithm improves a similar result due to Reif [R-85]. We provide an algorithm to compute the prefix sums of n numbers, each of $\log n$ bits, using $O(n)$ operations and $O(\log n / \log^{(2)} n)$ time on the CRCW PRAM. Later, we mention how to achieve the same result for numbers of $O(\log n)$ bits each, as well. This result is tight. That is, any algorithm using a polynomial number of processors must use this much time. This follows from the lower bound of [H-86] for circuits together with the general simulation result of [SV-84] between PRAMs and circuits. A direct proof was given recently in [BH-87].

[1] The base of all logarithms in the paper is 2.

Our new prefix sums algorithm employs a routine called the *Main* routine. The Main routine employs a routine called the *Basic* routine. We describe the Basic and Main routines and the prefix sums algorithm in this order.

The Basic routine.

Input: m numbers, each of m bits, and $m \cdot 2^{m^2}$ processors.

The Basic routine finds all m prefix sums in $O(1)$ time. Prior to an application of the Basic routine we must apply a routine called the *Precomputation* routine. The Precomputation routine initializes a table, called the *Precomputation table*, which is then used in the Basic routine.

Remark 2.2.1: In all applications of the Basic routine m will have the same value and therefore the Precomputation routine has to be performed only once throughout the prefix sums algorithm. It might be useful to know at this stage that $m \leq \log^{1/3} n$. We also note that the assumption used for our PRAM model is that a single processor can handle a word of $O(\log n)$ bits in $O(1)$ time.

We first describe the Precomputation routine.

The Precomputation routine. There are 2^{m^2} different possible inputs for the Basic routine. Due to the nature of the Basic routine, we call each set of possible inputs a *set of hypothetical inputs*. The domain of the Precomputation table has an entry for each set of hypothetical inputs. The range of each entry contains all the m prefix sums of the entry. The Precomputation routine simply applies the standard parallel prefix sums algorithm in parallel to each entry of the Precomputation table. It should be clear that this requires a total of $O(m \cdot 2^{m^2})$ operations and $O(\log m)$ time.

We return to the Basic routine. We provide a set of m processors for each of the 2^{m^2} sets of hypothetical inputs. The task, for each set of processors, is to discover if the actual inputs are identical to its hypothetical inputs. If they are not identical, the set of processors has completed its task. If they are identical, then the ith processor from the set simply looks up the ith prefix sum and outputs it in $O(1)$ time. We determine if the actual and hypothetical inputs are equal, as follows. Each processor in the set is responsible for one of the hypothetical inputs: the actual input is checked against its hypothetical input. If they are unequal the processor writes a 'fail' message to a memory location for its set of processors. Thus, only if there is no fail message in this memory location are the actual inputs identical to the hypothetical inputs for this set of processors. There should be exactly one set of processors for which the actual and hypothetical inputs are identical; this is the set of processors

that proceed to output the prefix sums. We have left to the reader the trivial details of how to assign the processors to their jobs. So the Basic routine requires a total of $O(m2^{m^2})$ operations and $O(1)$ time.

The Main routine.

Input: y numbers, each of $\log^{1/3}y$ bits and $y \cdot 2^{\log^{2/3}y}$ processors.

Remark 2.2.2: In all applications of the Main routine y is ("considerably") less than n. Specifically, y is $n \log^{(2)}n/(4 \log^{2/3}n \, 2^{\log^{2/3}n} \log n)$, as implied by Step 3 of the prefix sums algorithm given later.

The Main routine finds the prefix sums of these y numbers. Since the Main routine will apply the Basic routine it also has a precomputation step. We simply set m to be $\log^{1/3}y$ and compute the Precomputation table as in the Precomputation routine. Next, we provide a recursive description of the principal part of routine Main. Specifically, we describe the first and second recursive step. The second recursive step represents a general recursive step of the algorithm. The first recursive step is more degenerate.

The first recursive step of the Main routine: We divide the y numbers into groups of $\log^{1/3}y$ numbers, each of $\log^{1/3}y$ bits. We provide each group with $\log^{1/3}y \cdot 2^{\log^{2/3}y}$ processors. The prefix sums for each group are computed in $O(1)$ time, using the Basic routine; we call them *local* prefix sums. Consider only the sums of the numbers in each group (they are $\frac{y}{\log^{1/3}y}$ in number). The second recursive step solves the prefix sums problem with respect to these sums and the resulting prefix sums are called *global* prefix sums. For each of the y numbers, the first recursive step ends by computing its prefix sum as follows. We add its local prefix sum to the global prefix sum of the preceding group. The first recursive step takes $O(1)$ time.

The second recursive step of the Main routine: The input for the second step consists of the sums of the numbers in each group from the first step. This input has $\frac{y}{\log^{1/3}y}$ numbers, each containing at most $2\log^{1/3}y$ bits; it will be convenient to add bits so that each number has exactly $2\log^{1/3}y$ bits. We divide each of the numbers into two numbers of $\log^{1/3}y$ bits each: the leading and trailing $\log^{1/3}y$ bits, respectively. We then compute the prefix sums for each of the leading and trailing bits. We describe how to compute the prefix sums for the trailing bits. (The computation for the leading bits is identical.) This prefix sums computation is similar to the first recursive step, and is as follows. We divide the $y/\log^{1/3}y$ numbers into $y/\log^{2/3}y$ groups of $\log^{1/3}y$ numbers, each number of $\log^{1/3}y$ bits. We provide each group

with $\log^{1/3}y \cdot 2^{\log^{2/3}y}$ processors. The (local) prefix sums with respect to each group are computed in $O(1)$ time, using the Basic routine. The (global) prefix sums with respect to the $\frac{y}{\log^{2/3}y}$ sums of the groups are computed by the third recursive step. For each of the $y/\log^{1/3}y$ trailing numbers, we finally compute its prefix sum by adding its local prefix sum to the global prefix sum of the preceding group. There is one small issue which did not arise in the first recursive call. For each of the $y/\log^{1/3}y$ input numbers, we combine the two prefix sums of its trailing and leading numbers, in a further $O(1)$ time.

In order to evaluate the complexity of the Main routine we make the following two observations:

(1) The ith recursive step issues at most 2^{i-1} calls to the $i+1$st recursive step. (For instance, the second step issued one call for the third recursive step in the prefix sums computation of the trailing bits and another in the prefix sums computation of the leading bits).

(2) Consider a call for the ith recursive step. There are $\frac{y}{\log^{(i-1)/3}n}$ input numbers in each such call.

This leads to the following two conclusions:

(1) The depth of the recursion is $O(\log y/\log^{(2)}y)$. Since each recursive step takes $O(1)$ time, the Main routine runs in $O(\log y/\log^{(2)}y)$ time.

(2) The number of processors used for the first recursive step is $y\ 2^{\log^{2/3}y}$. We note that the number of input elements for each successive recursive call decreases by a factor of $\log^{1/3}y$, and the number of calls increases only by a factor of (at most) two. Therefore, the number of processors for the first recursive step suffices for implementing each of the later steps in $O(1)$ time. We conclude,

Theorem 2.2.1. The Main routine uses $O(\log y / \log^{(2)}y)$ time and $y\ 2^{\log^{2/3}y}$ processors.

The new prefix sums algorithm.

Input: n numbers, each of $\log n$ bits.

The prefix sums algorithm computes the n prefix sums of these numbers in time $O(\log n / \log^{(2)}n)$, using $O(n)$ operations, that is, using $n \log^{(2)}n / \log n$ processors. The algorithm has four steps.

Step 1: Each processor sequentially adds together $\log n / \log^{(2)}n$ input numbers.

Step 2: Add together sets of $x = 4 \log^{2/3}n \; 2^{\log^{2/3}n}$ numbers output by Step 1, using x processors per set, in time $O(\log x) = O(\log^{2/3}n)$. (Use the standard parallel prefix sums algorithm.)

Let y be $n \log^{(2)}n/(4 \log^{2/3}n \; 2^{\log^{2/3}n} \log n)$. It is easy to verify that (for large enough n) $2\log^{1/3}y \geq \log^{1/3}n$. The description below assumes that this inequality holds.

Step 3: Divide each number, output by Step 2, into (at most) $4 \log^{2/3}n$ pieces, each of $\log^{1/3}y$ bits, namely the trailing $\log^{1/3}y$ bits, the next least significant $\log^{1/3}y$ bits, and so on. For $1 \leq i \leq 4 \log^{2/3}n$, compute the prefix sums for the set of ith pieces using the Main routine, above. There are $4 \log^{2/3}n$ prefix sum subproblems, each of size $n \log^{(2)}n/(4 \log^{2/3}n \; 2^{\log^{2/3}n} \log n)$. Each subproblem uses $y \; 2^{\log^{2/3}y} \leq y \; 2^{\log^{2/3}n}$ processors. Thus, we need $n \log^{(2)}n \; / \log n$ processors and $O(\log n \; / \log^{(2)}n)$ time for this step.

Step 4: For each number input to Step 3, combine the $4 \log^{2/3}n$ prefix sums computed for that number (one per piece). This uses $O(\log^{2/3}n)$ time and $n \log^{(2)}n/(4 \log^{2/3}n \; 2^{\log^{2/3}n} \log n)$ processors. Finally, we "backtrack" through Steps 2 and 1 to compute the prefix sums for each number input to Step 1. The complexity of the backtracking is dominated by the complexity of Steps 1 and 2.

Overall, the four steps require $O(\log n \; / \log^{(2)}n)$ time and $O(n)$ operations.

Remark 2.2.3. It is easy to extend the above algorithm to numbers consisting of $c \log n$ bits each, for any fixed integer $c > 0$. We divide each number into c pieces, each of $\log n$ bits, namely the trailing $\log n$ bits, the next least significant $\log n$ bits, and so on. For $1 \leq i \leq c$, compute the prefix sums for the set of the ith pieces using the above parallel algorithm. For each number combine the c prefix sums computed for this number.

We conclude,

Theorem 2.2.2: The prefix sums of n numbers, of $O(\log n)$ bits each, can be computed in $O(\log n \; / \text{loglog } n)$ time using $n\text{loglog } n \; / \log n$ processors.

3. An Optimal 2-ruling Set Algorithm in $O(\log n/\text{loglog } n)$ Time

Subsection 3.1 reviews the deterministic coin tossing technique which was introduced in [CV-86a]. Subsection 3.2 gives the new 2-ruling set algorithm, where this technique is used.

3.1. The Deterministic Coin Tossing Technique

We illustrate the deterministic coin tossing technique by using it to break the (apparently) symmetric situation that arises in the r-ruling set problem.

In order to demonstrate our basic technique we give an $O(1)$ time algorithm using n processors for the $\lceil \log n \rceil$-ruling set problem. The algorithm is given for the EREW PRAM. Later we present a repeated application of the technique. It leads to an $O(1)$ time algorithm using n processors for the $\lceil \text{loglog} n \rceil$-ruling set problem.

Assumptions about the input representation: The vertices are given in an array of length n. The entries of the array are numbered from 0 to $n-1$. The numbers are represented as binary strings of length $\lceil \log n \rceil$. We refer to each binary symbol (bit) of this representation by a number between 0 and $\lceil \log n \rceil - 1$. The rightmost (least significant) bit is called bit number 0 and the leftmost bit is called bit number $\lceil \log n \rceil - 1$. Each vertex has a pointer to the next vertex in the ring (representing its outgoing edge). For simplicity we assume that $\log n$ is an integer.

Here is a verbal description of an algorithm for the $\log n$-ruling set problem. The algorithm is given later. Processor i, $0 \leq i \leq n-1$, is assigned to entry i of the input array (for simplicity, entry i is called vertex i). It will attach the number i to vertex i. So, the *present* "serial" number of vertex i, denoted $SERIAL_0(i)$, is i. Next, we attach to vertex i a new serial number, denoted $SERIAL_1(i)$, as follows. Let i_2 be the vertex following i. (That is (i,i_2) is in E). Let j be "the index of the rightmost bit in which i and i_2 differ". Processor i assigns j to $SERIAL_1(i)$. A remark in [CV-86a] explains how to compute j in $O(1)$ time even if there is no single instruction to do it in the repertoire of instructions of a processor.

Example. Let i be ...010101 and i_2 be ...111101. The index of the rightmost bit in which i and i_2 differ is 3 (recall the rightmost bit has number 0). Therefore, $SERIAL_1(i)$ is 3.

Next, we show how to use the information in vector $SERIAL_1$ in order to find a $\log n$-ruling set.

Fact 1: For all i, $SERIAL_1(i)$ is a number between 0 and $\log n - 1$ and needs only $\lceil \text{loglog} n \rceil$ bits for its representation. For simplicity we will assume that loglog n is an integer.

Let i_1 and i_2 be, respectively, the vertices preceding and following i. $SERIAL_1(i)$ is a local minimum if $SERIAL_1(i) \leq SERIAL_1(i_1)$ and $SERIAL_1(i) \leq SERIAL_1(i_2)$. A local maximum is defined similarly.

Fact 2: The number of vertices in the shortest path from any vertex in G to the next (vertex that provides a) local extremum (maximum or minimum), with respect to $SERIAL_1$, is at most $\log n$.

Observe that several local minima (or maxima) may form a "chain" of successive vertices in G. Requirement (1), in the definition of an r-ruling set, does not allow us to include all these local minima in the set of selected vertices. Our algorithm exploits the alternation property (defined below) of vector $SERIAL_1$ to overcome this problem.

The alternation property: Let i be a vertex and j be its successor. If bit number $SERIAL_1(i)$ of $SERIAL_0(i)$ is 0 (resp. 1), then this bit is 1 (resp. 0) in $SERIAL_0(j)$. (For $SERIAL_1(i)$ is the index of the rightmost bit on which $SERIAL_0(i)$ and $SERIAL_0(j)$ *differ*.)

Suppose that i_1, i_2 ... is a chain in G such that $SERIAL_1(i)$ is a local minimum (resp. maximum) for every i in the chain. Then:

Fact 3: For all vertices in the chain $SERIAL_1$ is the same (i.e., $SERIAL_1(i_1) = SERIAL_1(i_2) = \ldots$). (By definition of local minimum).

Below, we consider bit number $SERIAL_1(i_1)$ of $SERIAL_0$ for all vertices in the chain.

Fact 4: The following sequence of bits is an alternating sequence of zeros and ones.

Bit number $SERIAL_1(i_1)$ of $SERIAL_0(i_1)$,
bit number $SERIAL_1(i_2)$ $(= SERIAL_1(i_1))$ of $SERIAL_0(i_2)$, ...,
bit number $SERIAL_1(i_j)$ $(= SERIAL_1(i_1))$ of $SERIAL_0(i_j)$, ...

(This is readily implied by the alternation property.)

We can now understand why the technique is called deterministic coin tossing. We associated zeros and ones with the vertices, based on their original serial numbers; these serial numbers were set deterministically. This association allows us to treat (apparently) similar vertices differently. Finally, note that coin tossing can be used for similar purposes.

We return to the algorithm. We select the following subset of vertices.

We select all vertices i that are local minima and satisfy one of the following two conditions:

(1) Neither of i's neighbors (the vertices adjacent to i) is a local minimum.

(2) Bit number $SERIAL_1(i)$ is 1.

We say an unselected vertex is *available* if neither of its neighbors was selected and it is a local maximum. We select all available vertices i that satisfy one of the following two properties.

(1) Neither of i's neighbors is available.

(2) Bit number $SERIAL_1(i)$ is 1.

The selected vertices form a $\log n$-ruling set. Requirement (1) is satisfied since we never select two adjacent vertices. Requirement (2) is satisfied by **Fact 2** and since every local extremum is either selected or is a neighbor of a vertex that was selected.

Less informally we write the algorithm as follows. (Later, we will refer to this as the basic step.)

for Processor i, $0 \leq i \leq n-1$, **pardo** (perform in parallel)

$SERIAL_0(i) := i$

$SERIAL_1(i) :=$ "the minimal bit in which $SERIAL_0(i)$ differs from $SERIAL_0$ of the following vertex"

if $SERIAL_1(i)$ is a local minimum with respect to the two neighbors of i

then if either of the following is satisfied:

(1) neither of the vertices adjacent to i is a local minimum

(2) bit number $SERIAL_1(i)$ of $SERIAL_0(i)$ is 1

then select i

if neither i nor any of its neighbors were selected and if $SERIAL_1(i)$ is a local maximum with respect to the two neighbors of i

then (** i is *available*, and **) **if** either of the following is satisfied:

(1) neither of the vertices adjacent to i is available

(2) bit number $SERIAL_1(i)$ of $SERIAL_0(i)$ is 1

then select i

We have shown:

Theorem 3.1.1: A $\log n$-ruling set can be obtained in $O(1)$ time using n processors.

Below, we show how to repeat once more the basic step in order to find a $\log\log n$-ruling set.

In order to prepare the input for the second application of the basic step, we "delete" from G the vertices that were selected in the first application, their neighbors, and the edges incident to any vertex being deleted.

The input for the second application of the basic step is the remaining graph and vector $SERIAL_1$. $SERIAL_1$ will play the role played above by $SERIAL_0$ and a new vector $SERIAL_2$ will play the role of $SERIAL_1$. The degree of each vertex in the input graph is at most 2 (if

the directions of the edges are ignored). It is very simple to extend the basic step to handle vertices whose degree is ≤ 1. Vertices whose degree is 2 are treated as in the basic step (unless they have a neighbor whose degree is 1). The second application of the basic step will be as follows. (For an explanation see Fact 5 below).

for processor i, $0 \leq i \leq n-1$, **pardo**
  if vertex i or one of its neighbors have been selected
    in a previous application of the basic step
  then "delete" vertex i and the edges incident to it
for processor i, $0 \leq i \leq n-1$, such that i is in the remaining graph **pardo**
  case 1 $deg(i) = 2$
    then compute $SERIAL_2(i)$
      if the degree of each of i's two neighbors is 2
      then apply the basic step to i
  case 2 $deg(i) = 0$
    then select i
  case 3 $deg(i) = 1$
    then if either of the following is satisfied
      (1) the degree of i's neighbor is 2
      (2) i's neighbor is its predecessor
      then select i

The following fact helps to clarify the operation of the second application of the basic step.

Fact 5: Let i, j be adjacent in the input graph for the second application. Then:

$SERIAL_1(i) \neq SERIAL_1(j)$. (If they were equal each of them had to be a local maximum or local minimum at the first application. The selection of the ruling set implies that each local maximum or local minimum v is either selected or has a neighbor that is selected. Therefore, v must have been deleted and cannot be included in this input graph).

Fact 6: It is easy to deduce that the output graph consists of simple paths each comprising at most loglog n vertices. (Again, we assume for simplicity that loglog$n$ is an integer).

The vertices that were selected form a loglog$n$-ruling set. We have shown:

Theorem 3.1.2: A loglog$n$-ruling set can be obtained in $O(1)$ time using n processors.

If our original input is a directed path of n vertices, rather than a ring, we obtain a loglog$n$-ruling set by applying the basic step 2 times, as above.

Remark 3.1.1: It is interesting to mention that [CV-86a] show how to apply repeatedly the basic step to obtain a 2-ruling set in $O(\log^* n)$ time using n processors.

3.2. The New 2-ruling Set Algorithm

The algorithm presented in this section is based on a few changes to the optimal logarithmic time 2-ruling set algorithm of Section 2.3 in [CV-86a].

A high-level description.

Step 1. Find a $\log n$ / loglog n-ruling set. Apply the basic step of the deterministic coin tossing techniques twice, as described above, to get a loglog n-ruling set, which is in particular a $\log n$ / loglog n-ruling set. This takes $O(n)$ operations and $O(1)$ time.

Next, we consider only vertices v, which were not selected for the ruling set in Step 1. Recall that Step 1 associates the number $SERIAL_2(v)$ with vertex v, where $0 \leq SERIAL_2(v) < \log \log n$, and no two adjacent vertices have the same $SERIAL_2$ number. Below, we describe how to add more vertices to the $\log n$ / loglog n-ruling set to produce a 2-ruling set. These additional vertices are selected using the numbers $SERIAL_2$ associated with each vertex, as follows.

Step 2.

for $i = 0$ to $\log n$ / loglog $n - 1$ **do**
for each vertex v for which $SERIAL_2(v) = i$ **pardo**
if v is not in the ruling set and neither of the neighbors of v is in the ruling set
then add v to the ruling set

At each of its $\log n$ / loglog n "rounds", Step 2 selects a set of non-adjacent vertices. When Step 2 is finished, any vertex that was not selected must have a selected vertex as a neighbor. Thus this algorithm selects a 2-ruling set, as we wanted.

It remains to show how to implement Step 2 in $O(\log n$ / loglog $n)$ time using n loglog n / $\log n$ processors in order to conclude that these time and processor bounds hold for the whole 2-ruling set algorithm. We describe this implementation in three substeps:

Substep 2.1 We sort the vertices by their $SERIAL_2$ number. The outcome of this sort is that each vertex v will be given a number $RANK(v)$, $1 \leq RANK(v) \leq n$. No two vertices will have the same $RANK$.

Substep 2.2 For each v, $RANK(v) := RANK(v) + i n\log\log n / \log n$, where $i = SERIAL_2(v)$.

Substep 2.3 Implement the high-level description of Step 2 in $2\log n / \log\log n$ rounds (and not $\log n / \log\log n$ rounds). In round j $(1 \le j \le 2\log n / \log\log n)$, we process all vertices v such that $(j-1)n\log\log n / \log n < RANK(v) \le jn\log\log n / \log n$.

The implementation of Step 2 in Substep 2.3 guarantees that we never simultaneously process two vertices whose $SERIAL_2$ numbers are different.

Step 2.1 simply needs a bucket sort of n numbers in the range $[0, \log n / \log\log n - 1]$. The rest of this section shows how to perform such a sort in $O(\log n / \log\log n)$ time using $n\log\log n / \log n$ processors. It uses the new parallel prefix sum algorithm.

The sort proceeds in three stages (which are actually subsubsteps). In Stage 2.1.1, we count, for each number i, the number of vertices v for which $SERIAL_2(v) = i$. In Stage 2.1.2, using a sequential prefix sum algorithm, we count the number of vertices v for which $SERIAL_2(v) < i$, in $O(\log n / \log\log n)$ time. In Stage 2.1.3, for each vertex v, we determine a unique value $RANK(v)$. No two vertices get the same $RANK$.

Stage 2.1.1 proceeds in two substages. In substage 2.1.1.1, we divide the vertices into groups of size $\log n / \log\log n$. For each group, in $O(\log n / \log\log n)$ time, using one processor per group we count the number of vertices v for which $SERIAL_2(v) = i$, $0 \le i < \log n / \log\log n$. (We also determine, on the fly, for each vertex v, how many vertices w, preceding v in the group, satisfy $SERIAL_2(w) = SERIAL_2(v)$.) We obtain $n\log\log n / \log n$ sets of $\log n / \log\log n$ counts, one set per group. In Substage 2.1.1.2, using the new parallel prefix sum algorithm (or rather, $\log n / \log\log n$ of them), for each number i, we sum the $n\log\log n / \log n$ associated counts (for each i, one count per group). Clearly, this stage, implemented with $n\log\log n / \log n$ processors, uses $O(\log n / \log\log n)$ time.

Stage 2.1.2 is straightforward. In Stage 2.1.3, for each vertex v, we compute $RANK(v)$ using a single processor and $O(1)$ time. $RANK(v)$ will be: one, plus the number of vertices u such that $SERIAL_2(u) < SERIAL_2(v)$ (computed in Stage 2.1.2), plus the number of vertices w such that $SERIAL_2(w) = SERIAL_2(v)$ and w appears before v in the input array. The last number is obtained by adding the number of such vertices w that appear in groups prior to the group of v and the number of such vertices w that appear prior to v in its own group. Both numbers were computed in the first stage.

It now follows that the algorithm for bucket sort, with $\log n$ / loglog n buckets, uses $n$loglog n / log n processors and $O(\log n$ / loglog $n)$ time. We conclude

Theorem 3.2.1: A 2-ruling set can be obtained in $O(\log n$ / loglog $n)$ time using $n$loglog n / log n processors.

4. Optimal List Ranking in $O(\log n)$ Time

The list ranking algorithm given below is similar to the Basic List Ranking algorithm of Section 4 in [CV-86a]. There are two changes: we find a 2-ruling set using the new algorithm of the previous section and we use the new parallel prefix sums algorithm of Section 2 above. Section 4.1 gives the standard parallel list ranking algorithm. Section 4.2 gives the new list ranking algorithm.

4.1. The Standard Parallel List Ranking Algorithm

The standard parallel list ranking algorithm given below is due to [W-79]. Say that we have n processors. Assign a processor to each of the n elements. Denote the pointer of element i of the input array by $D(i)$ and initialize $R(i) := 1$, $1 \leq i \leq n$. We set $D(t) :=$ "end of list" (where t is the last element in the linked list), $D($ "end of list" $) :=$ "end of list" and $R($ "end of list" $) := 0$.

Iterate $\lceil \log n \rceil$ times:

for processor i, $1 \leq i \leq n$, **pardo**

$R(i) := R(i) + R(D(i)); D(i) := D(D(i))$ (To be called the short-cut operation, *performed* by i at $D(i)$).

Note that $\Omega(n \log n)$ short-cuts are made by this algorithm. It runs in time $O((n \log n)/p + \log n)$ using p processors and solves the list ranking problem, by placing the results in the vector R.

4.2. The New List Ranking Algorithm

Recall the recursive definition of the standard parallel prefix sums algorithm given in Subsection 2.1. Consider the following idealization of reality. Suppose we had been able to identify every second element in the linked list which is the input to our list ranking algorithm in $O(1)$ time using n processors. (Note that we mean every second element with respect to the list and not the input array containing the list.) This would have implied the following recursive list ranking algorithm:

(1) Rank a list consisting of these second elements taking the initial distance between two successive elements to be two.

(2) Find the ranking of the "first" elements based on the ranking of the successor of each such element in the input list.

Like the standard parallel prefix sums algorithm, such a list ranking algorithm would have run in $O(\log n)$ time using an optimal $(n/\log n)$ number of processors. Unfortunately, however, we do not know how to find the list of second elements that efficiently. To understand the problem observe that knowing recursively the lists of second elements is the same as knowing the ranking of the elements of the linked list mod 2, mod 4, mod 8, ... This idealized algorithm guided the design of the randomized list ranking algorithms of [V-84]. It also guided [CV-86a] to identify the 2-ruling set problem and led to the design of the new parallel list ranking algorithm below.

Initialization: $m := n$. As in the standard deterministic algorithm, denote the pointer of element i by $D(i)$ and initialize $R(i) := 1$, $0 \leq i \leq n-1$.

The algorithm which is given later should be read together with the commentary below. The purpose of the **while** loop of the algorithm is to "thin out" the input linked list into a list of length $\leq n/\log n$. The input to each iteration of the **while** loop is a linked list of length m stored in an array of length m. Vector D contains, for each element, the next element in this linked list.

The purpose of Step 2 is to enter either the value 1 or the value 0 into $RULING(j)$, for each j, $0 \leq j \leq m-1$, so that those elements with $RULING(j) = 1$, $1 \leq j \leq m$, form a 2-ruling set of the directed graph. Step 2 uses the new algorithm of Section 3.2 for finding a 2-ruling set.

In Step 3 we shortcut, in parallel, over each j such that $RULING(j) = 0$. The resulting list will contain exactly those elements in the 2-ruling set, of which there are at most $m/2$. We make some further comments on the operation of this step.

(a) Each element j for which $RULING(j)=1$ (an element of the 2-ruling set) is followed by at least one and at most two elements for which $RULING$ is 0.

(b) Each element over which we perform a shortcut will remain with no incoming pointers. Such elements will be "deleted" in Step 4.

(c) The parameter t stands for the present time. (This parameter increases as the algorithm progresses.) The information in $OP(i,t)$ enables us, later on, to reconstruct the operation of processor i at time t. This is used in Step 6 to derive the final value of

$R(D(j))$ by subtracting the present value of $R(j)$ from the final value of $R(j)$. For this reason we preferred here to name the processors performing the operations rather than to use the framework of Brent's theorem.

Step 4 contracts the input array for the present **while** loop iteration into a new array that contains exactly those elements in the new linked list.

When we arrive at Step 5, the length of the linked list at hand is $\leq n/\log n$. Step 5 applies the standard parallel list ranking algorithm in order to find the ranking of each element in this linked list.

Step 6 extends the list rankings to all elements of the original linked list using the information in $OP(.,.)$.

$t := 1$; (t is the present time)

while $m > n/\log n$ **do**

Step 1. (Initialization for the present **while** loop iteration).

for $j, 0 \leq j \leq m-1$, **pardo**

$SERIAL_0(j) := j$

Step 2. Compute a 2-ruling set into vector *RULING*, using the algorithm of section 3.2.

From now on we specify for each instruction the processors that perform it. Suppose p processors are available. Processor $i, 1 \leq i \leq p$, is assigned to segment $[(i-1)m/p, \ldots, im/p - 1]$ of the array that forms the input to this **while** loop iteration. (For simplicity we assume that m/p is an integer. Otherwise, we could assign Processor i to the segment including all the integers in the half open interval $((i-1)m/p - 1; im/p - 1]$.)

Step 3.

for Processor $i, 1 \leq i \leq p$, **pardo**

for $j := (i-1)m/p$ **to** $im/p - 1$ **do**

if $RULING(j) = 1$

then $OP(i,t) := (D(j),j,R(j))$;

$R(j) := R(j) + R(D(j))$; $D(j) := D(D(j))$ (shortcut).

if $RULING(D(j)) = 0$

then $OP(i,t) := (D(j),j,R(j))$;

$R(j) := R(j) + R(D(j))$; $D(j) := D(D(j))$ (shortcut).

Step 4. Apply the new parallel prefix sums algorithm of Section 2.2 with respect to the vector RULING. As a result:

(1) $m := \sum_j RULING(j)$, and

(2) each element j with $RULING(j) = 1$ gets its entry number in a (contracted) array of length m containing the output linked list.

(This array is the input for the next iteration (if any) of the **while** loop.)

od

Let T be the last time unit for which an assignment into $OP(\ ,\ )$ was performed.

Step 5. Apply a simulation of the standard deterministic parallel algorithm by p processors to the current array.

Step 6.

for Processor i, $1 \leq i \leq p$, **pardo**

for $t := T$ **downto** 1 **do**

$R(OP(i,t).1) := R(OP(i,t).2) - OP(i,t).3$.

(Comment. $OP(i,t).k$, $k=1,2,3$, represent the fields of $OP(i,t)$. If $OP(i,t)$ is undefined, the instruction is interpreted to be a null operation. Also, recall Comment (c) in the verbal description of Step 3.)

Complexity

We start by evaluating the operation and time requirements of the algorithm (so, at present, we assume that we have an unlimited number of processors available). Later, we use Brent's theorem to derive processor and time bounds. Initialization requires $O(n)$ operations and $O(1)$ time. Let us focus on one iteration of the **while** loop.

Step 1 takes $O(m)$ operations and $O(1)$ time.

Step 2 takes $O(m)$ operations and $O(\log m \ / \ \text{loglog}\ m)$ time.

Step 3 takes $O(m)$ operations and $O(1)$ time.

Step 4 takes $O(m)$ operations and $O(\log m \ / \ \text{loglog}\ m)$ time.

So each iteration of the **while** loop takes $O(m)$ operations and $O(\log m \ / \ \text{loglog}\ m)$ time. Each such iteration results in a linked list whose length is $\leq 1/2$ the length of the list when the iteration started. Therefore, after $O(\text{loglog}\ n)$ iterations we get a list whose length is $\leq n/\log n$. Summing up the operation and time complexity of the **while** loop gives $O(n)$ operations and $O(\log n)$ time.

Step 5 takes $O(n)$ operations and $O(\log n)$ time.

Step 6 requires the same number of operations and time as all the iterations of Step 3, since it follows its "footsteps".

So we got a total of $O(n)$ operations and $O(\log n)$ time. Applying Brent's theorem we get $O(n/p)$ time using any number $p \leq n/\log n$ of processors. We know that any such result can be alternatively stated as $O(\log n)$ time using $n/\log n$ processors. We leave the reader to

verify that the implementation problems as per Remark 2.1.1 following Brent's theorem can be readily overcome. We have shown:

Theorem 4.2.1: The list ranking problem can be solved in time $O(n/p)$ using $p \leq n/\log n$ processors.

References

[AM-86] R.J. Anderson and G.L. Miller, "Optimal parallel algorithms for list ranking", extended abstract, 1986.

[BH-87] P. Beame and J. Hastad, "Optimal bounds for decision problems on the CRCW PRAM", *Proc. 19th ACM Symp. on Theory of Computing*, 1987, to appear.

[CV-86a] R. Cole and U. Vishkin, "Deterministic coin tossing with applications to optimal parallel list ranking", *Information and Control* 70 (1986), 32-53.

[CV-86b] R. Cole and U. Vishkin, "Approximate and exact parallel scheduling with applications to list, tree and graph problems", *Proc. 27th Symp. on Foundations of Computer Science*, 1986, 478-491.

[CV-86c] R. Cole and U. Vishkin, "The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time", TR 52/86, Dept. of Computer Science, Tel Aviv Univ., 1986; also, TR 242, Dept. of Computer Science, Courant Institute, NYU, 1986.

[FL-80] M. Fisher and L. Ladner, "Parallel prefix computation", *JACM*, 27 (1986), 831-838.

[H-86] J. Hastad, "Almost optimal lower bounds for small depth circuits", *Proc. 18th ACM Symp. on Theory of Computing*, 1986, 6-20.

[MR-85] G. Miller and J. Reif, "Parallel tree contraction and its application", Proc. 26th Symp. on Foundations of Computer Science, 1985, 478-489.

[R-85] J. Reif, "An optimal parallel algorithm for integer sorting", *Proc. 26th Symp. on Foundations of Computer Science*, 1985, 496-503.

[SV-84] L. Stockmeyer and U. Vishkin, "Simulation of parallel random access machines by circuits", *SIAM J. Comput.* 13 (1984), 409-422.

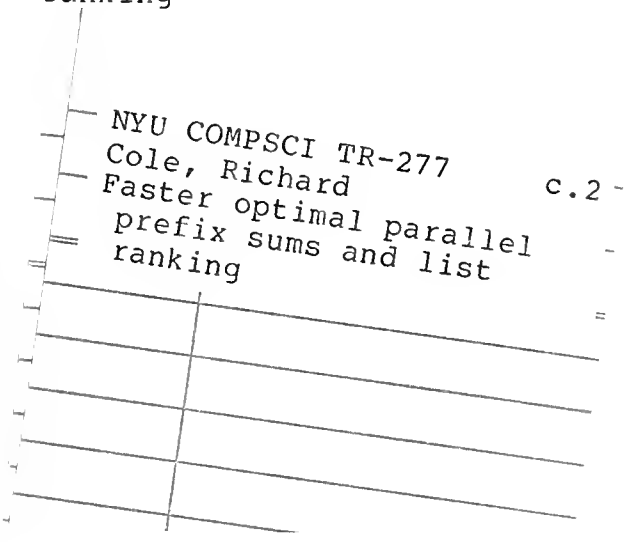
[TV-85] R.E. Tarjan and U. Vishkin, "An efficient parallel biconnectivity algorithm", *SIAM J. Comput.* 14 (1985), 862-874.

[V-83] U. Vishkin, "Synchronous parallel computation - a survey" , TR 71, Dept. of Computer Science, Courant Institute, New York University, 1983.

[V-84] U. Vishkin, "Randomized speed-ups in parallel computation", *Proc. 16th ACM Symp. on Theory of Computing*, 1984, 230-239. For a journal version of this paper see "Randomized speed-ups for list ranking", *J. Parallel and Distributed Computing*, to appear.

[V-85] U. Vishkin, "On efficient parallel strong orientation", *Information Processing Letters* 20 (1985), 235-240.

[W-79] J.C. Wyllie, "The Complexity of Parallel Computation", Ph.D. thesis, TR 79-387, Dept. of Computer Science, Cornell Univ., Ithaca, NY, 1979.


NYU COMPSCI TR-277
Cole, Richard
Faster optimal parallel
prefix sums and list
ranking
c.2